最終回

SystemVerilogシミュレーション 活用チュートリアル 03

第3回 アサーション・ベース検証手法

森田栄一

デバイスの記事　関連データ

本連載では，市販シミュレータの評価版を操作しながら，SystemVerilogを利用した検証手法について習得してきた．今回は，アサーションを使った検証手法について解説する．設計データ中にアサーション・モニタを埋め込んでおけば，デバッグの効率が上がる．アサーションをベースにしたカバレッジを取得することもできる．なお，ここで使用するシミュレータ（Questa）の評価版とサンプル設計は，本誌付属のCD-ROMに収録されている．（編集部）

今回の連載では，設計検証言語であるSystemVerilogについて，実際にシミュレーションとデバッグを体験するという観点から，その機能と利用方法について紹介しています．第3回の今回は，SystemVerilogアサーション（SVA）を用いたアサーション・ベース検証手法について解説します．アサーションを組み込む対象としてインターフェース（キーワードinterfaceで宣言）も導入します．

● アサーション・ベース検証とは

本連載の第1回でも触れましたが，アサーションなどの概念そのものが特別に斬新というわけではありません．ただ，これまでは標準的な実装方法がなく，現実には波形モニタ・ベースのデバッグ手法が幅広く普及していました．波形モニタ・ベースのデバッグは，回路の動作をその入り口と出口で観測するところから始まります．バグが何もなければこの手法は非常に効率的です．内部がどうであれ，回路は期待どおりに動作しているわけですから，論理的には何の問題もありません．もちろん，LSIとしての動作速度などは別の問題ですが…．

一定以上の大きさのプログラムには，かならずバグがあります．同じように回路の設計データもかならずバグを含んでいます．これが，期待する動作と異なった結果を出力します．**図1**のようなブロック構成でポイントAにバグがあるとします．このとき，テストベンチのパターンがポイントAにイベントを与えると，次のポイントBへ誤った結果を伝えます．その後，C→Dとその誤動作が伝播していきます．最終的に監視している出力ピンのところで期待値とのミスマッチが見つかります．ここからデバッグの開始です．出力ピンからD→C→B→Aとたどってバグを見つけます…，というようにうまくいけば良いのですが，そう簡単にはいきません．なぜなら，出力までの経路は一つとは限らないからです．実際にAにたどり着くまでには周辺

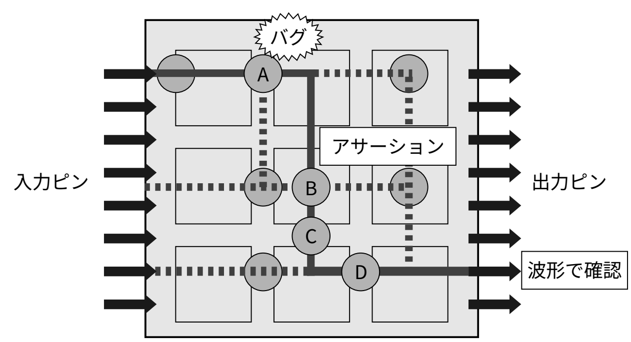


図1　アサーション・ベース検証の概要

ポイントAのバグが後段のB，C，Dと誤動作を伝播し，出力ピンに到達したところでバグを発見．ところが，出力ピンから直接Aを推定できないため，デバッグが複雑になる．ポイントBにアサーション・モニタがあれば，より早くAに到達できる

KeyWord　SystemVerilog，Questa，アサーション，デバッグ，アサーション・ベース検証，ディレクティブ，インターフェース，キャッシュ，カバレッジ，Open Verification Library

の回路に惑わされながらデバッグを進めていきます．その経路が長い場合には，非常に多くの時間を必要とします．

アサーション・ベース検証では，このポイントAまたはBにアサーション・モニタ（アサーションを自動監視するポイント）を埋め込みます．このモニタは，回路が期待しない動作を行った場合に，発火（こういう言いかたをする．“fire”とも言う）します．実際には，シミュレーション時のメッセージやデバッガの表示として確認できます．もし，ポイントAに指定したアサーションであれば，その場でバグを検出できます．もちろん発火したアサーションがかならずしも原因のポイントというわけではありませんが，回路の中に一定量のアサーションをしかけておくことで，問題が検出された場所からバグのポイントまでの経路が短くなり，デバッグの効率を上げることができます．

このような説明は，アサーションの有効性についての説明としてどこかで聞いたことがあるかと思います．しかし，実際の波形ベースのデバッグでも，回路の内部情報をいっさい見ないわけではありませんし，ある程度注意するべきポイントは確認しているはずです．それでもアサーションが便利な理由は，目視で波形を追いかけるのではなく，ツールが自動的に監視してくれること，長期サイクルにわたる多数の信号変化（シーケンス）を監視できることなどです．アサーション・ベース検証を用いると，目視では不可能なレベルのチェックを実行できます．

また，**図2**のようにブロック・レベルのテストベンチについては問題なくpassした設計も，システム・レベル検証では周辺回路からの予期せぬ入力によって正しく動作しない場合があります．このとき，周辺回路側にバグがある場合と，ブロック・レベル検証で想定していたパターンが不足していた場合の二つの原因が考えられます．これらのいずれの場合でも，ブロック・レベルで挿入されたアサーションが発火することで，すばやく問題点へ到達できます．

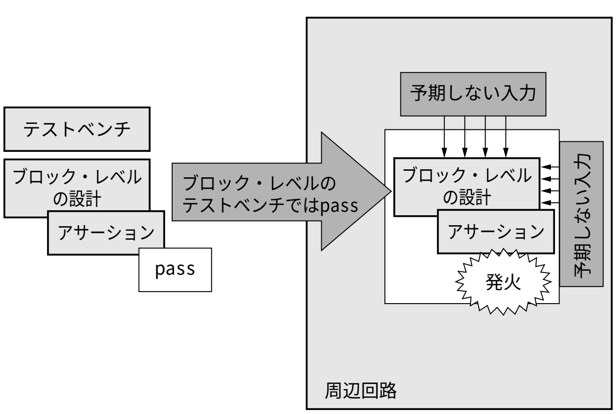


図2　システム検証時のアサーション
ブロック・レベルのテストベンチをpassした場合でも，システム・レベルでは単体テストで予期できなかった入力が発生する場合がある．アサーションを利用すると，該当ブロックが想定していない入力が存在することを即座に判定できる

● 時系列の信号変化を動作仕様として定義

アサーション・ベース検証では，時系列の各信号変化を動作仕様として定義します．例えば，「あるサイクルで信号aが‘1’となった3サイクル後にbが‘1’になるというシーケンスが検出された場合，その次のサイクルではcが‘1’でなければならない」という仕様があったとします（**図3**）．SystemVerilogアサーションではプロパティ（キーワード`property`で宣言）として次のように表記します．

```
property ab;
@(posedge clk) a ##3 b |=> c;
endproperty
```

このプロパティ記述には次の要素が含まれています．

- **@(posedge clk)によるサイクル定義** ── クロックの立ち上がりごとにサイクル・カウントされ，すべての信号値がサンプリングされる
- **##3によるサイクル・カウント** ── aとbの間には，3サイクル分の遅延があることを意味する．aとbのシーケンスが定義されている
- **|=>インプリケーション演算子** ── 先行する（左辺）シーケンス（条件）が成立した場合に，プロパティのチェックを行うことを意味する．|=>の場合には，右辺の第1サイクルは左辺の最終サイクルから1サイクル後にな

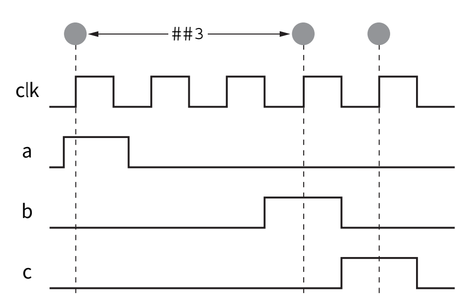


図3　アサーションで定義するシーケンス
クロックの立ち上がりごとに各信号の値をサンプリング．aの3サイクル後にbが，さらに1サイクル後にcが成立

る(ノン・オーバラップ). |-> が使用された場合は，左辺の最終サイクルと右辺の第1サイクルは同じサイクルになる(オーバラップ).

このプロパティ記述のチェックを行うには，キーワードassertを用いてシミュレータに検証対象としての指示を行います．例えば，次のように記述します．

```
assert property (ab);
```

assertのようなキーワードを「ディレクティブ(指示子)」と呼びます．ほかのディレクティブにはcoverなどがあります(coverについては後述する).

● SystemVerilogのインターフェース機能を用いる

アサーションを組み込むためのしくみとして，System Verilogのインターフェース機能を用います．インターフェースを用いると，モジュール間の接続仕様を独立に定義して管理できるようになります(**図4**)．サンプル設計の例を用いて解説します．

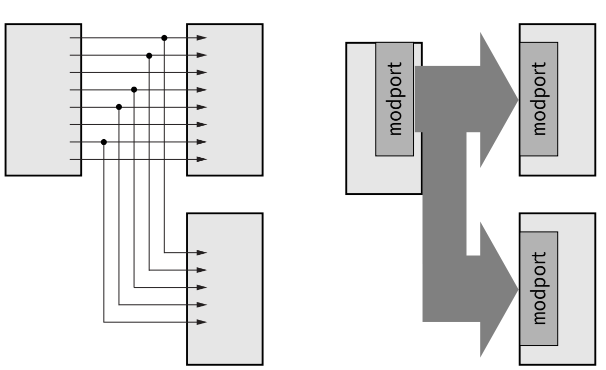
まず，今回使用する設計データを本誌付属のCD-ROMからコピーします．今までと同じように，C:\exampleにコピーします．今号のCD-ROMには，連載第1回から第3回までのサンプル・データをすべて収録しています．第3回のディレクトリからコピーするように注意してください．また，CD-ROMにはシミュレータ「Questa」の最新版6.1eも収録しています．READMEファイルに従って準備を行ってください．

最初にtop.svを参照します(**リスト1**)．インターフェース「pmbus」を宣言しています．最初に，このインターフェースを使用するすべてのモジュールが参照するクロック信号を宣言しています．次に，インターフェース内のローカル信号，さらにmodport宣言を使用して，接続するモジュールから参照できる信号を指定しています．

top.svでは，インターフェースpmbusをpm0という名称でインスタンスしています．このpm0がインターフェースの実体となります．次にmemory.svとcache.svの内容を確認します．

モジュールmemoryのポートには，pmbus.bus_m m0という宣言しかありません(**リスト2**)．インターフェースpmbus中のmodport bus_mをm0という名まえでインスタンスしてポートとしています．同じようにモジュールcacheでは，pmbus.bus_pをp0としてインスタンスしています．cacheでは，プロセッサと接続される側の信号はインターフェースを用いずに，通常のポート宣言となっています．

top.svに戻ってモジュール・インスタンスの部分を確認します．二つのモジュールmemoryとcacheのポートm0,



(a) 従来のポート接続　(b) インターフェースを用いた接続

図4　インターフェース

個々のピン仕様はインターフェース内に宣言され，各モジュールとはmodportを通して接続される．ピン仕様が変更された場合も，各モジュールではポート仕様を変更する必要がない．modportを使用せずにインターフェースと全ピンを接続することも可能

リスト1　インターフェース宣言

```
interface pmbus (input bit CLK);   ← クロック宣言
  logic [7:0] ADDR;
  wire [15:0] DATA;
  wire RW;
  wire STRB;
  wire RDY;

  modport bus_p (                  ← プロセッサ側ポート
    input CLK,
    inout DATA,
    output ADDR, RW, STRB,
    input RDY);

  modport bus_m (                  ← メモリ側ポート
    input CLK,
    inout DATA,
    input ADDR, RW, STRB,
    output RDY);

    <<中略>>

endinterface

module top;                        ← モジュールtop

    <<中略>>

  // Memory side interface
  pmbus pm0 (.CLK(clk));           ← インターフェースのインスタンス

    <<中略>>

  memory m (.m0(pm0));             ← インスタンシエーション

    <<中略>>

endmodule
```

リスト2　モジュールmemory

```
module memory(pmbus.bus_m m0);  ← ポート宣言はmodportのみ

<<中略>>

  always @(posedge m0.CLK) if (m0.STRB == 0) begin
    i = m0.ADDR;
    if (m0.RW == 1) begin          ← m0.***でmodportにアクセス
      repeat (4) @(posedge m0.CLK);
      data_r = mem[i];
      rdy_r = 0;
      @(posedge m0.CLK)
      data_r = 'bz;
      rdy_r = 1;

<<以下略>>
```

リスト3　アサーション記述

```
/*********** 三つの同じアサーション **********/

sequence RdStrb;                              } ← シーケンス宣言
  !STRB ##1 STRB && RW;
endsequence
                                                プロパティ宣言
property RdRdy;                               } ←
  @(posedge CLK) RdStrb |=> STRB[*0:8] ##0 !RDY;
endproperty

assert property (RdRdy);  ← assertディレクティブによる指定

a_01: assert property (RdRdy);  ← ラベル付きの宣言

assert property (@(posedge CLK) RdStrb
|=> STRB[*0:8] ##0 !RDY);  ← 1行ですべて宣言

/*********** そのほかのアサーション **********/

property StrbOnce;
  @ (posedge CLK) !STRB |=> STRB;
endproperty
assert property (StrbOnce);

property DataValid;
  @ (posedge CLK) (!STRB && !RW) |=> !$isunknown(DATA);
endproperty                                  ↑
assert property (DataValid);          組み込み関数の利用

property BusIdle;
  @ (posedge CLK) !RDY |=> RDY && (DATA === 16'hzzzz);
endproperty
assert property (BusIdle);
```

p0に対して，トップ・レベルでインスタンスしたインターフェースpm0をマッピングしています(**リスト1**).

さらにmemoryとcacheの各モジュール内のインターフェースへの入出力は，m0.DATA，p0.STRBのようにインスタンス名とmodport名によって構成されています(**リスト2**)．例えば，モジュールcache内でp0.STRBをドライブした場合，memory内では，m0.STRBでその値を参照することができます.

通常の接続では，モジュール間の接続仕様が変更された際には，すべてのモジュールのポート宣言とインスタンシエーション記述を書き換える必要があります．インターフェースを用いると，その仕様が変更された場合にも，インターフェース宣言を変更するだけです．もちろん，各モジュール内のmodportに対する入出力記述の変更は必要になります.

この例のように，modportを使用してインターフェースに対する入出力のコンポーネントを明示する方法は，バス・インターフェースに対するマスタ，スレーブを指定するような場合に便利です．インターフェースの使用方法としては，modportを用いずに全ピンを接続することも可能です.

● インターフェース中にアサーションを記述

インターフェースは再利用可能な接続仕様の宣言です．この中にアサーションを記述しておくことで，接続した各モジュールがインターフェースを利用する際のルールを規定して監視することができます．実際にインターフェース記述の後半にアサーションを定義しています(**リスト3**)．それぞれのアサーションについて簡単に解説します.

1) property RdRdy

シーケンスRdStrbは，リード・サイクルの開始を定義しています．プロパティRdRdyでは，リード・サイクルの開始を条件にインプリケーションの右辺側シーケンスを監視します．右辺側では，STRB == 1(インアクティブ)が0～8サイクル繰り返される間にRDY信号がアクティブになることを期待しています．ここで，STRB信号のインアクティブ条件を指定しているのは，RDY信号がアクティブになる前に再びSTRB信号が発行されることを禁止しているためです.

2) property StrbOnce

STRB信号は1サイクルのみアクティブになるという仕様を定義しています.

3) property DataValid

ライト・サイクルの開始を条件に，次のサイクルでDATAが正しく出力され，不定値を含まないことをチェックしています．不定値の確認には，$isunknownという組み込み関数を利用しています.

4) property BusIdle

RDY信号が発行されてバス・サイクルが終了した後に，DATA信号が解放(ハイ・インピーダンス状態)されることをチェックします.

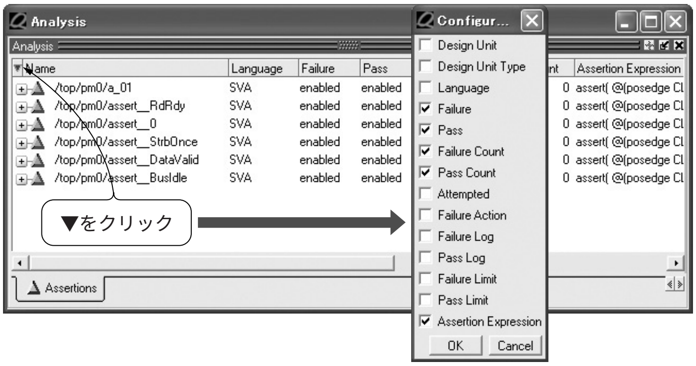


図5　Assertionsタブ
Analysisウィンドウの一つのタブとして表示される．アサーション情報の表示と動作の設定を行うことができる

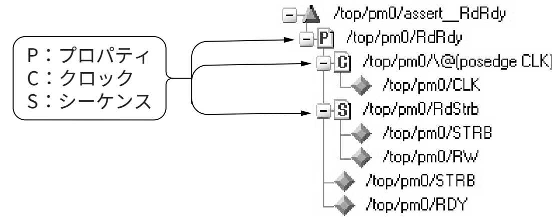


図6　アサーションの展開表示
アサーションを展開すると，関連づけられた信号，およびアサーションを構成するプロパティやシーケンスを確認することができる

このインターフェースを用いた場合，これらのアサーションは自動的にチェックされます．このようなアサーションの使用方法は，バスに接続されるデザイン・コンポーネントのプロトコル・モニタとして利用できます．

● デバッグ情報が表示されるように設定する

アサーションを挿入した設計データに対して，シミュレーションを実行してみましょう．Questaを起動してC:\exampleディレクトリに移動します．ライブラリの作成，コンパイル，ローディングを実行します．以下のコマンドを打ち込んでください．

```
QuestaSim> cd c:/example
QuestaSim> vlib work
QuestaSim> vlog *.sv
QuestaSim> vsim -assertdebug top
```

-assertdebugオプションによって，アサーション・ベース検証に必要なデバッグ情報をすべて表示することが可能となります．バッチ・モードでアサーションのfailのみをログで監視するような場合，本オプションは不要ですが，今回のチュートリアルではすべてのシミュレーションにこのオプションを使用します．

● Analysisウィンドウを表示する

シミュレータ起動後，次のコマンドを実行します．

```
VSIM> view assertion -undock
```

Analysisウィンドウで，Assertionsタブ（**図5**）が表示されます．本連載の第2回では，Analysisウィンドウ中のCovergroupsタブを利用しました．Assertionsタブでは，宣言されたアサーションとそれぞれに対する設定を行うことができます．Workspaceで選択した階層以下に含まれるアサーションが表示されるので，すべてのアサーションを表示するために最上位階層（top）を選択します．

アサーションの表示フォーマットを確認するために，top.sv内の最初の三つのアサーションに注目します．これらのアサーションはまったく同じ内容を定義しています（**リスト3**）．それぞれのアサーションは次のように表示されています．

- **プロパティを宣言した後にassert記述** ── `assert_`<プロパティ名>
- **assertにラベルを付ける** ── <ラベル名>
- **assert記述のみで構成** ── `assert_`<通し番号>

`assert`ディレクティブのみで宣言した場合には，通し番号のみが表示されアサーションの内容がわかりにくくなります．いったんプロパティを宣言するか，ラベルを付けることを推奨します．アサーション内の情報を確認するために，`assert_RdRdy`を展開表示します（**図6**）．ここで，「C：クロック」，「P：プロパティ」，「S：シーケンス」と関連づけられた信号群を確認できます．

次に，ウィンドウ内の表示情報を確認します．Assertionsタブの▼（下向きの三角）をクリックすると，表示情報を設定するダイアログボックスが現れます（**図5**）．下記の項目にのみチェックを入れます．

- **Failure**　：failチェックのenable/disable状態
- **Pass**　：passチェックのenable/disable状態
- **Failure Count**　：fail回数
- **Pass Count**　：pass回数
- **Assertion Expression**　：アサーション記述

設定後，スクロールしてAssertion Expressionを確認します（必要に応じて表示枠を調整する）．ここでは，シーケ

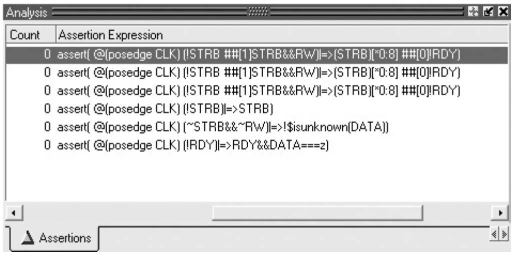


図7　Assertion Expression

参照しているシーケンスなどをすべて展開して表示する．ソース・コードを参照するよりもアサーションの内容を確認することが容易である

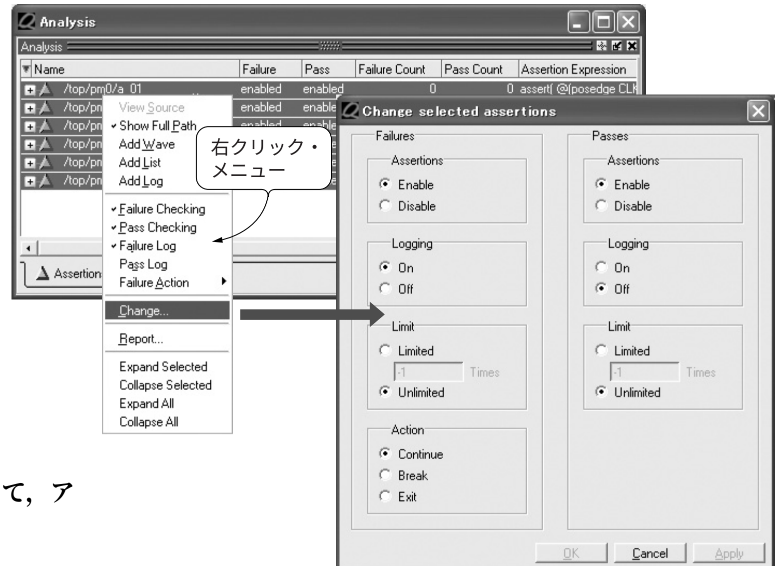


図8　アサーションの動作設定

すべてのアサーションに対して，その動作を設定する．特定のアサーションを選択することで，任意のアサーションの動作のみを変更したり，無効にしたりすることが可能

ンスの内容などをすべて信号レベルの記述に展開して，アサーションを確認できます(**図7**)．

シミュレーションを実行する前に，アサーションの動作を指定します．Assertionsタブですべてのアサーションを選択した状態でマウスを右クリックし，メニューから「Change」を選択します(**図8**)．表示されたダイアログ・ボックスでは，下記の設定を確認します．

● **Failures**

Assertions　：Enable
Logging　：On
Limit　：Unlimited
Action　：Continue

● **Passes**

Assertions　：Enable
Logging　：Off
Limit　：Unlimited

アサーションのenable/disable制御は重要な機能の一つです．アサーションそのものは，シミュレータに実装された専用エンジンで高速に実行できますが，通常のシミュレーションに比べて実行時間は長くなります．そのため，検証の終了したアサーションは無効化(disable)する必要があります．ソース・コードでアサーション記述をコメント・アウトすることでも無効化できますが，再コンパイルが必要で，手順が煩雑になります．

● シミュレーションを実行する

シミュレーション・パターンには，本連載の第2回で使用したランダム・テスト・パターンを用います．キャッシュ内部の記述を変更しているため，サイクル数が異なりますが，出力メッセージなどの仕様はすべて同じです．

あらかじめexampleディレクトリ内に準備したwave.doを実行します．

```
VSIM> do wave.do
```

このスクリプトは，次の実行内容を含んでいます．

- 観測信号(最上位 /* とインターフェース内 /pm0/*)を追加
- アサーションをwaveウィンドウに追加
- waveウィンドウのカテゴリ分け
- waveウィンドウのアンドック

その後でシミュレーションを実行します．

```
VSIM> run -all
```

シミュレーション終了後，Assertionsタブを確認します．ここで最初に確認するのはpass/failともに0回のアサーションです．アサーションの場合，failしなければpassというわけではありません．例えば，下記のアサーションを例に考えます．

```
assert property (@ (posedge clk) A ##3
B| => C);
```

シミュレーション中，(A ##3 B)のシーケンスが一度も発生しなかった場合，プロパティのチェックは行われま

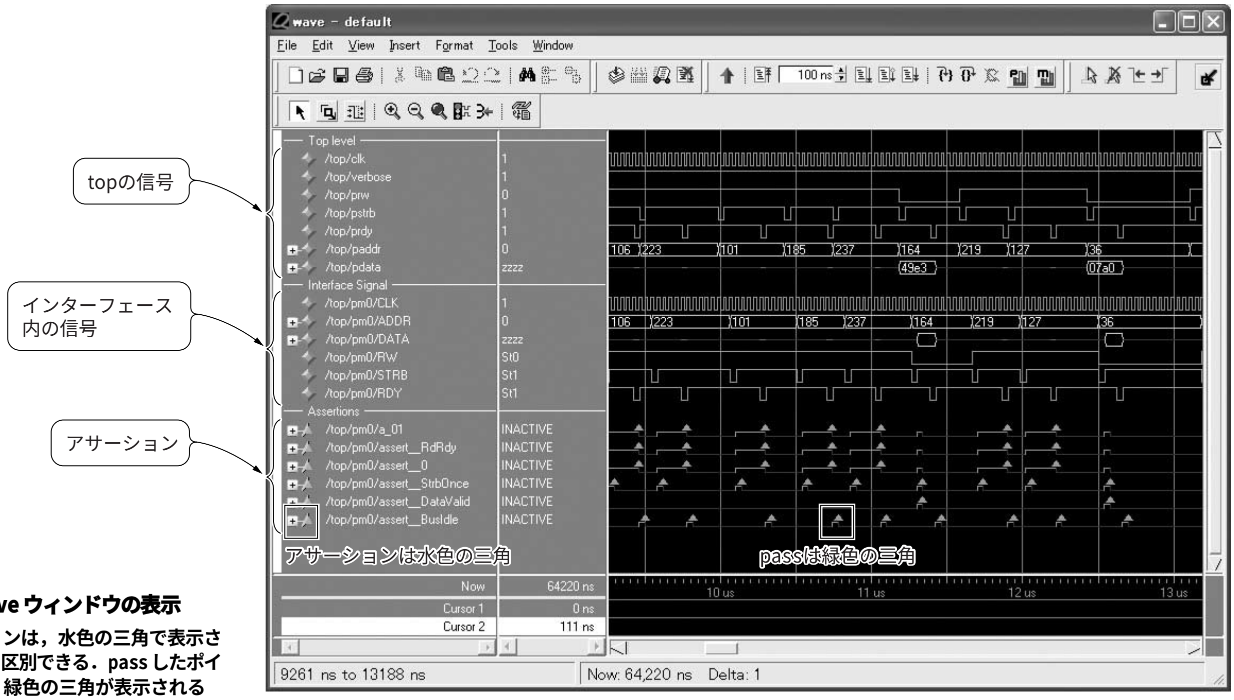


図9　wave ウィンドウの表示
アサーションは，水色の三角で表示され，信号と区別できる．pass したポイントには，緑色の三角が表示される

せん．したがって，fail はしませんが pass もしません．これは「設計として問題がない」ということを意味しません．「必要なチェックが行われていない」ということを意味します．このようなアサーションがチェックされるように，シミュレーション・パターンを追加したり，ランダム条件を変更したりすることで，アサーションの「カバレッジ」を引き上げる必要があります．今回のシミュレーションでは，DataValid の pass と fail の両方が 0 になっていることを確認できます．これは，proc.sv 内のランダム制約の部分で `WRITE :=  0` として，ライト・サイクルが発生しないように指定されているからです．ここを書き換えて，リード・ライト・サイクルの発生比率を同一にします．proc.sv を編集後，リコンパイル，リスタートします．

```
VSIM> vlog -incr *.sv
VSIM> restart -f
VSIM> run -all
```

DataValid の pass/fail の数を確認します．

● wave ウィンドウで確認する

次に，wave ウィンドウで結果を確認してみましょう（**図9**）．wave ウィンドウ上では，アサーションは水色の▲（上向きの三角）で表示されて，ほかの信号との違いを確認できます．ここでも，Assertions タブと同じように＋（プラス）マークをクリックして展開することで，各アサーションに関連づけられている信号を確認できます．展開した各信号はそのまま波形を確認できるので，fail したアサーションのデバッグに有効です．wave ウィンドウでは，pass したタイミングで緑色の▲マークが，fail したタイミングで赤色の▼（下向きの三角）マークが表示されて，視覚的に結果を確認できます．このセッションではすべてのアサーションが pass するので，fail ポイントの表示は見ることができません．

● ライト・バッファを挿入して動作を確認する

本連載の第1回で解説しましたが，本サンプル設計のキャッシュはライト・スルー方式を使用しているため，ライト・ヒット・ミスにかかわらず，キャッシュとメモリの両方へ書き込みを行います．このことは，ライト・サイクルにおいては，キャッシュを使用しない場合と比べてシステムとしての性能が落ちてしまいます．これに対して，ライト・バッファを挿入することで，ライト・サイクルのサイクル数を減らすことができます．ライト・バッファの効果を確認するため，最初のセッション・シミュレーション時間を確認しておきます．64220ns で終了しているはずです．

ライト・バッファを挿入したバージョンのソース・コー

リスト4　top.svの修正

```
  cache_wb c (.paddr(paddr), .pdata(pdata),
// cache c (.paddr(paddr), .pdata(pdata),
```

cache側をコメント・アウト　　cache_wbのコメントを外す

リスト5　ライト・バッファ

```
/*************** Memory Access ***************/

always @(posedge p0.CLK) if (sysacc) begin
  while (membusy) @(posedge p0.CLK);
    maddr_r = paddr;
    mstrb_r = 0;
  if (prw == 1) begin

<<リード動作>>

  end else begin
    mrw_r = 0;
    ob_data = pdata;
    membusy = 1;
    prdy_r = 0;
    @(posedge p0.CLK);
    prdy_r = 1;
    assign mdata_r = ob_data;
    @(posedge p0.CLK);
    mstrb_r = 1;
<<以下略>>
```

membusyによる制御　　ライト中はmembusyを出力　　ライト動作

ドをcache_wb.svとして準備しています．すでに同一ディレクトリ内に収録しているので，*.svの対象となっています．モジュールcache_wbを利用するには，top.svのインスタンス部分を書き換える必要があります（**リスト4**）．ライト・バッファは，非常に簡単なアルゴリズムで実現しています．通常，ライト・サイクルはメモリからのrdy信号を待って終了しますが，ライト・バッファ付きのキャッシュでは，プロセッサからのstrb信号のすぐ後にrdy信号を返します．その際，データをバッファに書き込んで，バッファからメモリへの書き込みを実行します．バッファからの書き込み中もリード・ヒットのキャッシュからの読み出しは行うことができます．リード・ミスやライト・サイクルで発生するメモリ・アクセスは，バッファからの書き込み中はその実行を待たされることになります．この制御をmembusyという信号で行っています（**リスト5**）．

top.svを編集してシミュレーションを実行します．コンパイル後，リスタートします．

```
VSIM> vlog -incr *.sv
VSIM> restart -f
```

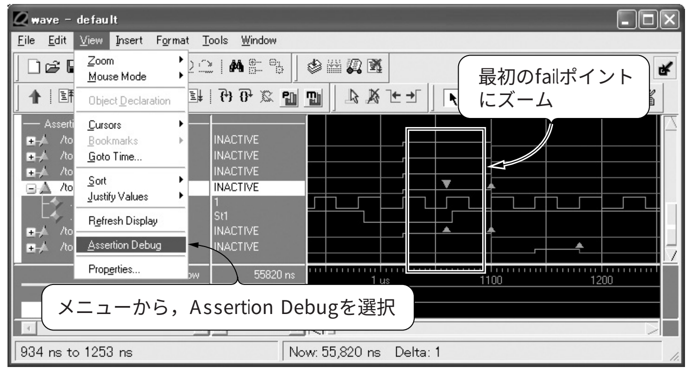


図10　failポイントを確認
ここからアサーションのデバッグがスタートする

次のコマンドを実行して，全信号をログに追加します．この作業は，アサーション・デバッグのために内部ノードをトレースする際に必要となります．

```
VSIM> add log -r /*
```

シミュレーションを実行します．

```
VSIM> run -all
```

ランダム・テストベンチの結果といっしょに，アサーションのfailログが出力されています．アサーションのメッセージをわかりやすくするために，テストベンチ側のメッセージを非表示にします．次のコマンドを実行して，シミュレーションを再実行します．

```
VSIM> restart -f
VSIM> force /verbose 0
VSIM> run -all
```

モジュールtopにverboseという変数を定義して，ランダム生成タスク内の$displayの実行を抑止しています．この機能は，今回のソース・コードから追加したものです．前回のサンプル・コードでは利用できません．

表示されたメッセージからアサーションのfailポイントが判断できます．同時にwaveウィンドウとAssertionsタブから，アサーションStrbOnceがfailしたことがわかります．StrbOnceは，「STRB信号が1サイクルのみアクティブになる」という簡単なものです．waveウィンドウとデバッグGUI（graphical user interface）を使用して，問題点を探していきます．

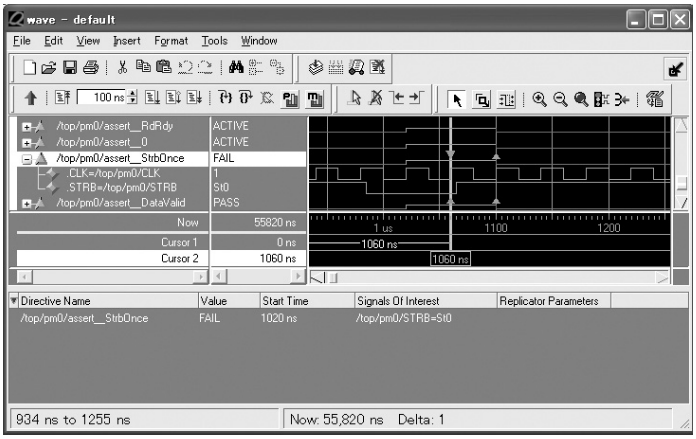


図11　Assertion Debugサブウィンドウ
このウィンドウでは，アサーションのfailポイントをクリックすることで，failの原因となっている可能性のある信号をリストアップしてくれる

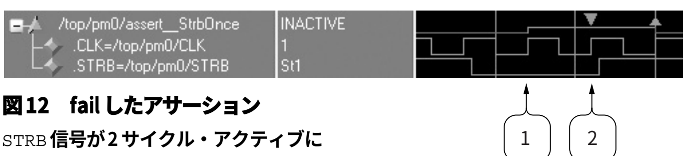
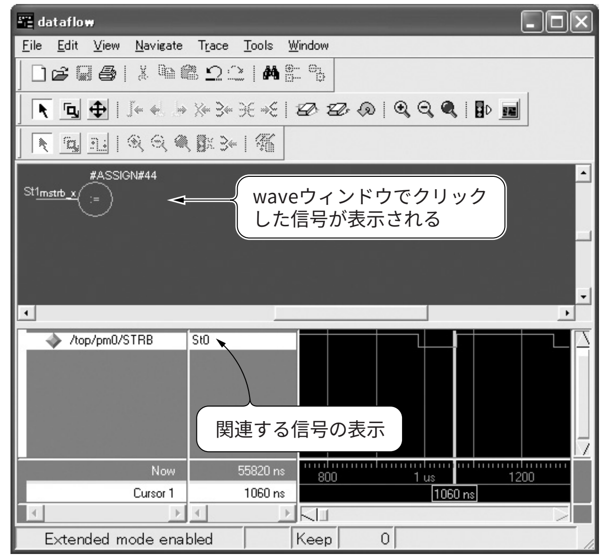
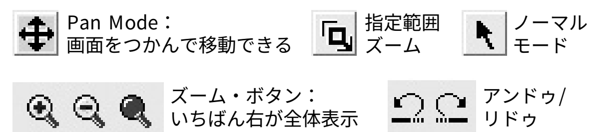


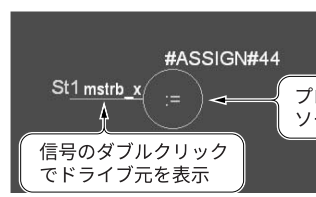
図12　failしたアサーション
STRB信号が2サイクル・アクティブになったところでfailしていることがわかる



（**a**）ウィンドウ全体



（**b**）アイコンの説明



（**c**）信号が表示されたエリア

図13　dataflowウィンドウ
信号と，信号をドライブするプロセスの関係を表示する．プロセスをクリックすることで，該当するソース・コードを表示する．階層構造や複数ソース・コードを意識せずにデータの流れだけをトレースできる

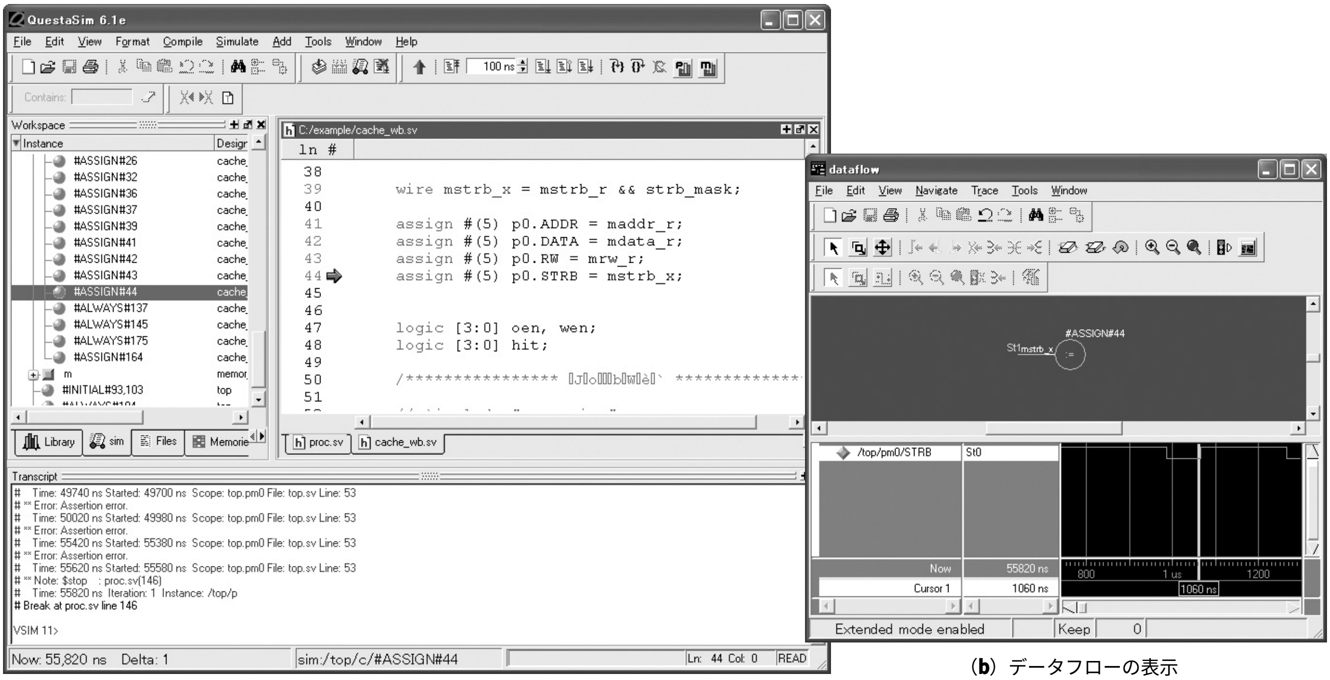
● failポイントを解析する

waveウィンドウ上でfailポイントを確認します．最初のfailポイントにズームします(**図10**)．通常のシミュレーションでは，最初にfailしたポイントでシミュレーションを停止してデバッグする方法が一般的です．2番目以降のfailは，事前のfailによって誤動作した可能性があるからです．failしたアサーションを展開して関連づけられた信号を確認します．

次に，waveウィンドウのメニューから「View」→「Assertion Debug」を選択します(**図10**)．waveウィンドウの下部にサブウィンドウが表示されます(**図11**)．このウィンドウは，failポイントでその原因と考えられる信号のリストを表示します．failの▼マークをクリックしてください．ウィンドウ内に信号名「STRB」が表示されます．展開した波形表示から，STRB信号が2サイクル出力された時点でfailしていることを容易に判断できます(**図12**)．

STRB信号のfailポイント近くでダブルクリックします．**図13**のdataflowウィンドウが表示されます．このウィンドウでは，波形から信号変化を起こしたソース記述へのトレースを行えます．**図14**のように，dataflowウィンドウとソース表示を同時に確認できるようにします．

データフローの最初の表示では，STRB信号とそれをドライブするassign文が○(円)で表されています．この○をクリックします．sourceウィンドウに該当するソース・コードが表示されて矢印でその場所が示されます(**図15**)．ここでは，STRB信号にmstrb_xという信号が単純に代入されていることがわかります．マウス・カーソルを「Pan Mode」に切り替えて，dataflowウィンドウに表示された内容を中心に持ってきます．**図13**のように，assign文の入力信号をダブルクリックします．前段の入力プロセスが表示されます．表示されたassign文をクリックしてソース・コードを確認します．次の代入文を確認できます(抜粋)．



（**a**）ソース・コードの表示

（**b**）データフローの表示

図14　データフローとソース・コード
dataflowウィンドウでは，ソース・コードをトレースしながらのデバッグとなるため，データフローとソース・コードの両方の情報を確認できる状態でデバッグすると便利

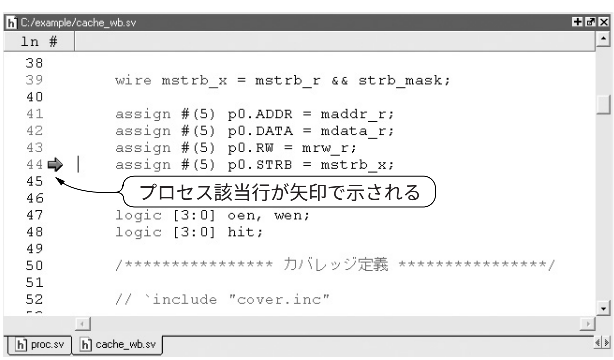


図15　sourceウィンドウ内のプロセス表示
dataflowウィンドウのプロセスからソース・コード中の代入文が表示される．これによって信号とその代入文の関係を迅速にトレースできる

```
mstrb_x = mstrb_r && strb_mask
```

mstrb_x信号は，mstrb_r信号とstrb_maskという信号の論理演算の結果であることがわかります．ここで，各信号のドライブ元を探査してもよいのですが，dataflowウィンドウでは，選択したプロセスの入出力信号を波形表示しています（**図16**）．この波形とソース・コードを同時に参照することで，問題となっている入力を判断できます．実際にfailしたポイント周辺では，strb_mask信号は'1'を維持しています．つまり，mstrb_r信号に問題があるようです．mstrb_r信号をダブルクリックします．**図17**の表示になります．ここでは，mstrb_rをドライブしている二つのプロセスが表示されます．一方は，initial文による初期化のプロセスのため問題はありません．もう一方のalways文をクリックしてソース・コードを確認します．

このalways文は，**リスト5**に示したライト・バッファの記述であることがわかります．また，prw信号の値からライト・サイクルであることがわかります．**リスト5**の「ライト動作」の最後の1行に注目してください．ここでmstrb_rに'1'を出力しています．ところが，この代入文の前に1クロック・サイクルのウェイトが入っています．このウェイトによって，mstrb_r信号だけがほかの信号より1サイクル遅れて変化していたのです．この1行を削除（コメント・アウト）することでエラーは取り除くことができます．実際にソース・コードを編集してシミュレーションを再実行してみてください．アサーションのfailがなくなることを確認できます．シミュレーションの総時間は55820nsとなるはずです．ライト・バッファがないときと比べて，性能をおよそ13％改善できたことになります．

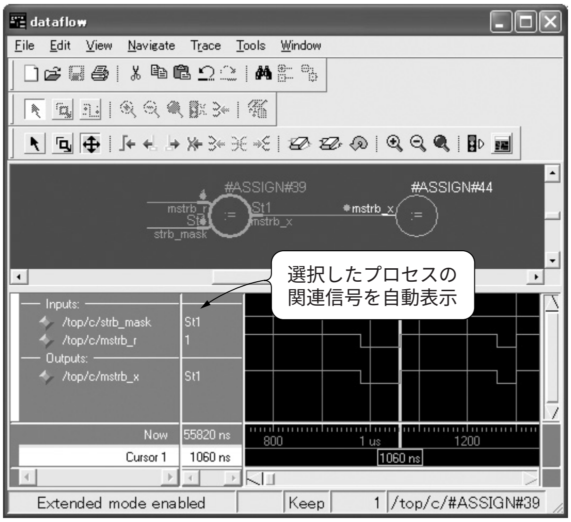


図16　選択プロセスの入出力信号の表示
プロセスを選択して表示したソース・コードとdataflowウィンドウ内の波形表示を併用することで，問題となる入力を容易に判断できる

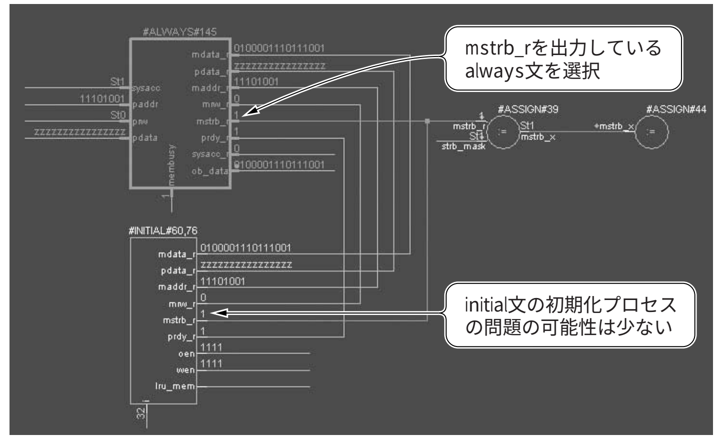


図17　問題となる信号をドライブしているalways文
最初のassign文（アサーションの発生ポイント）から3段のトレースで問題点に到達

このデバッグ作業の中で最初にAssertionsタブを確認した際に，StrbOnceではpassとfailの両方がカウントされていました．アサーションで定義した内容は単純なので目視でも簡単にバグを発見できそうな気がしますが，特定サイクルでのみエラーが発生するような場合は，エラーのポイントを発見するのは難しくなります．このような全サイクルで監視できることも，アサーションの大きなメリットと言えます．

このシミュレーション・セッションでは，発生したアサーションfailからデバッガGUIを使用したソース・デバッグの実際を体験できたかと思います．なお，strb_mask信号はデバッグの雰囲気を体験するために挿入した冗長な信号です．

今回のアサーションは，デザイン・コンポーネントが通信を行うためのルール（アクセスのタイミングなど）を定義しています．今回は触れていませんが，ブロック・レベルのデバッグ（キャッシュ・コントローラの動作アルゴリズム検証）もアサーションの重要な利用目的の一つとなります．

● アサーションを用いたカバレッジ

最後に，アサーション機能をベースにしたカバレッジについて少しだけ解説します．SystemVerilogアサーションの重要な機能がシーケンス・モニタであることはすでに述べました．このシーケンス・モニタを使用して，特定の動作（ファンクション）のカバレッジを取得することができます．第2回で紹介したカバレッジは，デザイン内がある一定の状態に達したかどうかを計測する「データ指向カバレッジ」でした．一方，SystemVerilogアサーションのシーケンスを用いたカバレッジは「コントロール指向カバレッジ」と呼ばれます．このカバレッジの取得には，coverディレクティブを用います．

top.sv内のインターフェース宣言の最後にcoverディレクティブを用いたライト・サイクルのカバレッジを記述してあります（**リスト6**）．すでにコンパイルされ，準備も整っているので，最後のシミュレーション・セッションからリスタートします．

```
VSIM> restart -f
```

リスト6　コントロール指向カバレッジ

```
/*********** coverディレクティブ **********/

sequence WrStart;
  !STRB && !RW ##1 STRB;
endsequence

sequence WrData;
  (!$isunknown(DATA) && STRB)[*0:8];
endsequence

sequence WrDone;
  !RDY ##1 RDY && (DATA === 16'hzzzz);
endsequence

sequence WriteCycle;
  @ (posedge CLK) WrStart ##1 WrData ##0 WrDone;
endsequence

cover property (WriteCycle);
```

（注釈）三つのシーケンスを宣言／三つのシーケンスからライト・サイクルを構成／ライト・サイクルの実行を監視

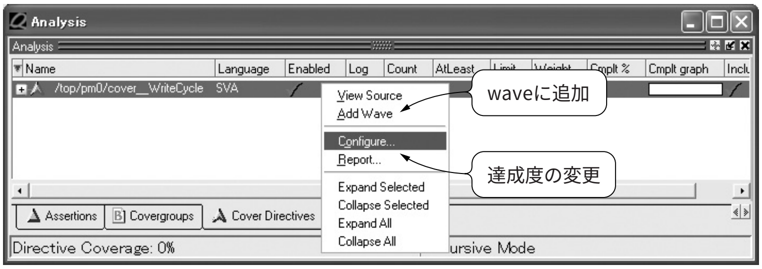


図18　Cover Directivesタブ
coverディレクティブによるカバレッジの表示と制御が行える

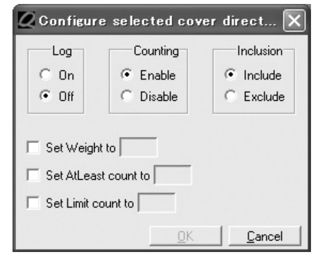


図19　カバレッジ達成度の変更
それぞれのカバレッジの重要度や達成回数の指標を指定できる

次のコマンドを実行します．

```
VSIM> view cover
```

Analysisウィンドウの Assertionsタブに追加して，Covergroupsタブと Cover Directivesタブが追加されます（**図18**）．Cover Directivesタブ上でカバレッジ Write Cycleを選択し，マウスを右クリックします．表示されるメニューから「Add Wave」を選択します．waveウィンドウにカバレッジが追加されます．coverディレクティブは，矢じりのようなアイコンで表示されます．シミュレーションを実行します．

```
VSIM> run -all
```

Cover Directivesタブでカバレッジを確認します．カバレッジは100％となっています．これはデフォルトのゴール（AtLeast）が1回になっているためです．第2回のcovergroupでは，言語の機能を用いて達成度の設定を変更しました．coverディレクティブでは，シミュレータの機能を用いて達成度や重み付けを変更できます．Cover Directivesタブでカバレッジを選択して，マウスを右クリックします．メニューから「Configure」を選択します．**図19**のダイアログボックスが表示されます．「Set Weight to」は重要度を指定します．「Set AtLeast count to」でゴール（目標回数）を指定します．これらの項目を指定して，カバレッジの表示がどのように変わるかを確認してください．

今回のシミュレーションと同時に第2回の演習で使用したカバレッジ（covergroup）を取得することもできます．covergroup記述は，cover.incというファイルに格納してあります．cache_wb.sv内のinclude文のコメント・アウトを外すことで利用可能になります．covergroupを追加することで，Analysisウィンドウのすべてのタブを使用することになります．

● アサーションのその先には…

アサーションを記述することで，効率的に検証できることを理解していただけたかと思います．サンプル設計で使用したアサーションは簡単なものばかりですが，実際の設計ではより複雑なアサーションを記述する必要が出てきます．どのように複雑なアサーションでも，シミュレーションを使用したデバッグの方法は，大きくは変わりません．

アサーション・ベースの検証では，複雑なアサーションを簡単に記述するために検証用ライブラリを用いることが一般化しつつあります．標準化されたもののうち，もっとも有力なものとして，OVL（Open Verification Library）と呼ばれるライブラリがあります．OVLはSystemVerilogアサーションでも記述されており，カバレッジをとることも可能です．本シミュレータでも，もちろん使用可能です．また，アサーションを静的に解析するフォーマル検証という方法もあります．数学的手法を用いて，特定のアサーションが論理的にエラーとならないことを完全に証明します．

このような新しい検証手法についても，機会があればご紹介したいと思います．

もりた・えいいち
メンター・グラフィックス・ジャパン（株）

<筆者プロフィール>
森田栄一．Questa（旧名V-System）にかかわり始めたころは20代だったのに，気づけば40代間近．HDL設計環境を見続けてきて感じるのは，その進化の遅さ．設計規模と速度は2けたも向上しているのに，検証手法はやっとSystemVerilogなどが登場してきたにすぎない．そんなに速くてもついていけないのかもしれないが….